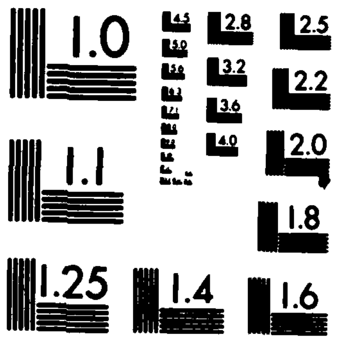
AD-A181 623   PC EXPERT SYSTEM DBMS INTERFACE TOOLS PHASE 1   1/1
FEASIBILITY STUDY(U) APPLIED LOGIC SYSTEMS SYRACUSE NY
K A BOWEN 03 DEC 86 DAAB10-86-C-0551

UNCLASSIFIED   F/G 12/5   NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(1)

PC Expert System / DBMS Interface Tools

Final Report

Contract DAAB10-86-C-0551
(SBIR Phase I Feasibility Study)

USACECOM Vint Hill Procurement Office
R&D Field Station Support
P.O. Box 1551, VHFS
Warrenton, VA 22186-5120

Requisition/P.O. Number:	W73DMT-0115-86
Effective Date of Contract:	14 May 1986
Expiration Date of Contract:	3 December 1986

Principal Investigator:
Kenneth A. Bowen
(315)-471-3900

Project Engineer:
Keith Hughes
(315)-471-3900

Applied Logic Systems Inc.
Box 90, University Station
Syracuse, New York 13210

Summary

The technical goals of this Phase I effort were to explore the feasibility of the construction of direct interfaces between Prolog and databases managed by conventional DBMS systems. The Prolog utilized was ALS Prolog and the DBMS system utilized was dBase III. Expert tools which assist programmers in the construction of interfaces between ALS Prolog and dBase III were successfully constructed and tested. This study demonstrates that it is quite feasible to easily construct interfaces between the languages used for the implementation of expert systems and the files managed by conventional database management systems.

Two further feasibility studies were conducted. The first was the feasibility of constructing expert systems which ease the task of preparing graphical presentations of data held in DBMS files. On the negative side, the study showed that it is not feasible to achieve this using conventional graphics presentation packages which are oriented towards interactive use. On the positive side, the study showed that it is feasible to couple such an expert system with lower-level programming language packages of graphics ~~presentation routines. This~~ will allow one to build an expert graphics presentation tool which can be interfaced to a variety of different databases. The final study concerned the feasibility of utilizing the database interfaces to construct tools for easy input of ASCII data into existing DBMS data files. This too was shown to be quite feasible by coupling the database interfaces to previously constructed tools for reading structured ASCII data into Prolog.

All three feasibility studies were quite successful and demonstrate that powerful, flexible, and easily used tools of the type described can be constructed for the PC environment.

1. Summary Results of the Phase I Work

The Phase I work was highly successful. The original technical objectives (from Section 4 of the Phase I proposal) were:

• **Objective 4.1.** Construct Prolog rules representing the manner in which dBase III generically organizes its datafiles, indexing files, and database definitions.

• **Objective4.2.** Construct low-level access functions which are able to navigate in dBase III data and index files and which can read individual fields of dBase III records.

• **Objective 4.3.** Construct a prototype collection of Prolog predicates which utilize the results of 4.1 and 4.2 to construct custom Prolog interfaces to a particular dBase III database. User-friendly aspects and full packaging as an expert assistant will be deferred in this feasibility study.

• **Objective 4.4.** Select a suitable commercial graphics presentation package for use with dBase III. Develop Prolog rules which describe the files it requires and the commands to invoke it properly. Develop a prototype collection of Prolog predicates which utilize these rules to create graphics presentations of information from the dBase III database. Again, user-friendly aspects and full packaging as an expert assistant will be deferred in this feasibility study.

• **Objective 4.5.** Applied Logic Systems has already developed methods of specifying the format of input data files and reading them into internal Prolog forms (under a current SBIR Phase I study for the National Institutes of Health -- see Section 4). Develop prototype Prolog predicates which utilize the interfaces constructed in 4.3 to load the input data into the dBase III database.

Our overall results of the Phase I effort can be summarized as follows:

- On objectives 4.1-4.3, we accomplished substantially more than was proposed.
- On objective 4.4, we encountered limitations due to the nature of almost all graphics presentation packages, but have devised approaches to overcome these limitations.
- On objective 4.5, we completely met the goal.

2. Details of Accomplishments by Objective

2.1 Objectives 4.1-4.3:

2.1.1. We constructed a small collection of very low-level predicates which were coded in C and which supplied primitive functions for database file access. These included predicates for:

- opening and closing arbitrary files
- locating at an arbitrary byte position in a file
- determining the current position in a file
- determining if end of file has been reached
- reading a specified number of characters
- writing a specified number of characters
- reading integers and reals from files
- writing integers and reals to files

These primitive predicates were originally added to ALS Prolog as new built-in predicates. This is a rather awkward process, and comsumes space in the standard Prolog which it is hard to justify. However, they no longer must be added as (non-standard) built-in predicates. ALS Prolog has been extended to support foreign predicates implemented in C (and later FORTRAN, Pascal, etc.). Consequently, the primitive predicates required for the database interfaces can be loaded as foreign C-coded predicates via the standard foreign program interface. The delivered version of the software will be the version incorporating these predicates as non-standard Prolog builtins (the original approach). However, during the Phase II effort, the interface will be configured to smoothly use the foreign program interface.

2.1.2. A set of Prolog predicates was implemented (in Prolog, in terms of the C-coded predicates of 2.1.1) to enable the reading and writing of the specific datatypes supported by dBase III, such as number, boolean, date, character, etc. (Padding and justification of fields is appropriately accounted for.)

2.1.3. A Prolog definition of the internal headers of dBase III datafiles was coded (utilizing 2.1.1 and 2.1.2 above). This definition can be used to direct the reading of information from the header of an existing file, or to create a new header for a new empty datafile.

2.1.4. A Prolog definition of the B+ - tree structure of dBase III indexing files was constructed. This definition directs the navigation through the indexing files for both reading and writing records in data files.

2.1.5 A small interface-construction expert program was implemented in Prolog. This program is used to construct the actual interfaces between Prolog and specific dBase III files. The expert conducts a short dialog with the programmer to obtain such information as the name of the target dBase data file, the names of any associated indexing files (which must be obtained from the programmer since dBase III does not support a uniform data dictionary), etc. The expert program then reads the header of the target data file, and afterwards writes Prolog code defining the virtual interface between Prolog and the data file. The code is stored in a file named by the programmer. In order to utilize the interface, the programmer simply loads this interface code file along with any other Prolog code defining the program which will manipulate the data from the database. The interface supports both reading from and writing to the dBase III datafiles. It also utilizes the indexing files for both reading and writing, and updates the indexing files whenever new records are written (by the interface) into the data file. An arbitrary number of such interfaces can be attached to a given Prolog program.

2.1.6. A large-scale geographic database was created as a collection of dBase III files (derived from a collection of Prolog files supplied with Borland's Turbo Prolog), and was interfaced (via the tool of item 2.1.5 above) to a simple natural language query program written in Prolog (also derived from a program supplied with Turbo Prolog). The test was quite successful.

2.1.7. A prototype high-level DBMS expert system was constructed for use by Applied Logic Systems programmers in constructing the interface tools for given DBMS systems. This DBMS expert has knowledge of the generic structure of files and indexing methods used by DBMS systems. It assists the user (a high level Prolog programmer) in constructing the analogues of the tools 2.1.3-2.1.5 above for a given (new) DBMS. After a moderate-sized dialog with the programmer, the DBMS expert writes definitions in the style of 2.1.3 & 2.1.4, and then writes information used by the interface expert for interfaces specific to the given DBMS (in the style of item 2.1.5 above).

Further details of the interfaces are presented in the attached Appendicies.

2.2 Objective 4.4:

We examined a number of standard commercial graphics presentation packages designed for use with DBMS systems such as dBaseIII. Hardly any of the available graphics presentation packages support command-line arguments or batch operation. Virtually all are oriented towards interactive use, which presents a problem for remote control by an expert program (no matter what language is used to implement the expert program). For those which utilize the standard DOS input and output, it would be possible to communicate with the graphics program through the temporary pipeline buffer files used by DOS. However, the effort required for this does not seem to be warrented. A much better approach appears to be to use the foreign program interface for Prolog in order to couple Prolog to a standard commercial collection of graphics presentation routines (many such collections exist). These can be utilized by an top-level expert presentation system coded in Prolog which also interfaces to the appropriate DBMS files.

The feasibility of this approach was tested using a small collection of graphics routines supplied with the Manx C compiler. The test was highly successful.

2.3. Objective 4.5:

We were successful in connecting the previously constructed routines for reading structured data from ASCII files with our routines for interfacing from Prolog to DBMS files for the purpose of inputting data to existing datafiles. The experience showed that such separate coupling introduces inefficiencies. However, our previous experience and the present work show that a better approach would be to modify our original approach so that the target for the ASCII file reader is not internal Prolog structure. Instead, the target for the ASCII file reader should normally be database file output through the very low-level routines coded in 2.1.1 under Objectives 4.1-4.3.

3. Evaluation and Next Steps

The evaluation of the feasibility studies for all of the technical objectives was highly positive. They demonstrate that each of the objectives can be achieved with efficient programs which are quite compact and suitable for the PC environment. Moreover, our tests of ALS Prolog on 286-class machines (e.g., IBM AT and accelerator boards for PCs), 386-class machines (e.g., COMPAQ 386 and accelerator boards for PCs and ATs), and 68000-class machines (e.g., Macintosh, SUN workstations, accelerator boards for PCs and ATs) show that the resulting programs will be extremely efficient and productive.

The immediate next steps are the following:

3.1. Extend the DBMS interfaces to other systems such as R:Base 5000, primarily by the route of extending the high-level DBMS expert (2.1.7 above). We have already obtained the cooperation of MicroRim for R:Base 5000 in this regard, and will approach other companies.

3.2. Push some of the Prolog-coded predicates down to C or assembler-coded predicates for improved efficiency (especially those dealing with aspects of indexing). This will be done after significant effort is carried out on 3.1. so as to choose an optimal set of predicates for recoding.

3.3. Select an appropriate collection of C or assembler-coded graphics presentation routines for interfacing to databases. This may be a commercially available set. However, we will expore the value of coding the set ourselves, since we might be able to tune it especially well to the needs of the interface.

3.4. Complete the recoding of the ASCII file input reader for inputting data into existing databases. This has begun and is expected to be quite routine given the experience with merging our previous work with the present concerns.

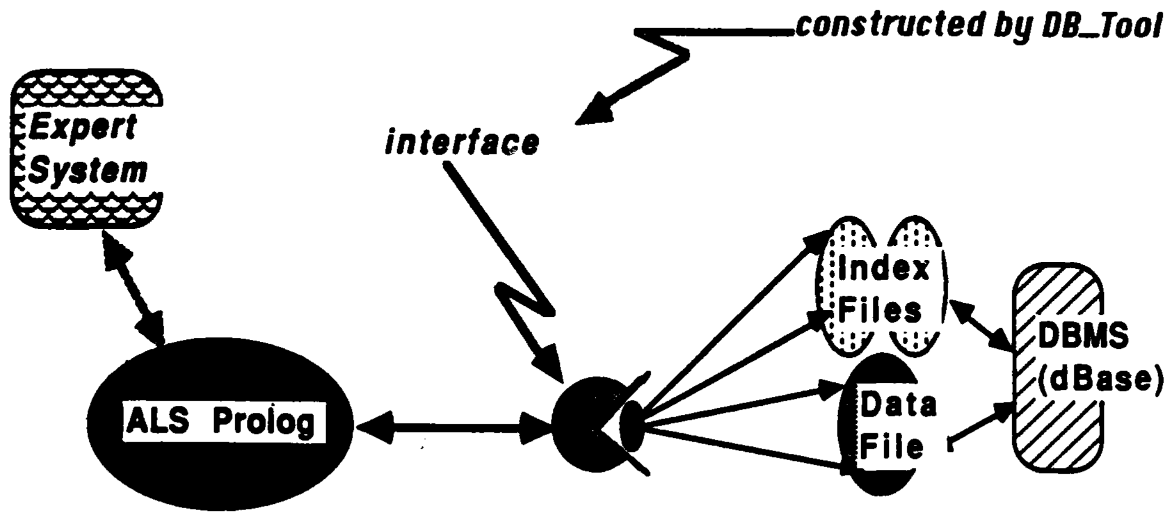
Appendix 1. Description of the Interface Tool.

Prolog ↔ dBaseIII Interface

Most expert systems applications require the system to access and manipulate extensive databases of information. In many cases, this information resides in a database developed under a standard DBMS, and is accessed and maintained by applications programs written in standard programming languages such as C or FORTRAN or specialized Database Programming Languages associated with the DBMS. However, these languages are not usually suitable for implementation of complex expert systems. Instead, Prolog or LISP or specialized expert shells (written in Prolog or LISP) are utilized. Because of the size of the database and supporting applications as well as its importance to the organization owning it, it is usually unthinkable to convert the database and the supporting applications to Prolog or LISP. Thus a gap exists between established databases and expert systems which must utilize the data residing in the databases.

Applied Logic Systems has developed a method of bridging this gap without disrupting the database and the other application programs making use of the database. The method uses the existing database to provide expert systems written in ALS Prolog with *virtual views* of the information in the database. This virtual view is defined by interface code (written in ALS Prolog) which is created by an expert system called Access Expert (which is itself written in ALS Prolog). Access Expert conducts a brief interview with the programmer, determining such things as the database and relations to be accessed, the nature of the access (fields or whole records, read or write), etc. DBTool also reads information from the DBMS data dictionary or from headers in data and index files. It then combines this information to synthesize the interface code, which it compiles and writes to a file for storage.

The over-all architecture of the resulting interface is suggested by the following sketch:



The construction of interfaces between Prolog and relational databases is especially natural because the fundamental notion of both is the notion of a relation. Abstractly, an n-place relation is a collection of n-tuples of objects. The relation is said to be 'true' for each n-tuple actually in the collection.

From the perspective of relational databases, a relation is a rectangular table. Each of the n columns of the table corresponds to one of the positions in the n- tuples, and the rows of the table correspond to the n-tuples belonging to the collection defining the relation. Many relational DBMS systems identify each table with a file containing the data in appropriate format. The name of the file is often identified with the relation name.

Prolog sees simple relations as collections of facts. Let 'parent' be the name of a 3-place Prolog relation. A fact for 'parent' is a statement that 'parent' holds of a certain 3-tuple of objects, namely (john, mary, ruth). This is usually expressing in the form

```
parent(john, mary, ruth).
```

This fact corresponds to the row

```
john     mary     ruth
```

in the relational database table for the relation. Since Prolog views relations as collections of facts, there is an obvious one-to-one correspondence between the Prolog facts and the tuples in the relational database table view of the relation. And at a higher level, there is a Prolog relation corresponding to each database table.

A Prolog <=> DB interface constructed by Access Expert allows a Prolog program to behave as if it actually contained the collection of facts corresponding to the tuples in the database table. However, this collection of facts never really exists in its totality in Prolog. Instead, individual Prolog instances of the appropriate facts are constructed "on the fly" by the interface as they are needed by the Prolog program. When Prolog finishes using such an instance, it is discarded. For example, suppose that PARENTS.DBF is a dBaseIII data file containing a 3-column table, and that 'parent' has been connected by the interface to PARENTS.DBF. The file PARENTS.DBF may contain 5,000 tuples, but only a few of the corresponding facts for 'parent' exist in the Prolog image at any one time. As the computation of the Prolog program requires instances of 'parent', the interface accesses PARENTS.DBF and constructs one or more appropriate instances of 'parent' from tuples contained in PARENTS.DBF. If dBaseIII indexing files have been constructed for PARENTS.DBF, the interface utilizes them in locating appropriate tuples from the PARENTS.DBF table.

The interface is two-way. ALS Prolog provides the programmer with a special built-in called `db_assert'. This is a database-oriented version of the standard assert. A Prolog call of the form

```
db_assert(p(john, mary, harry))
```

will cause the interface to write the corresponding tuple

```
john   mary   harry
```

into PARENT.DBF. The dBaseIII indexing files for PARENTS.DBF are not currently updated, but this facility is being added at present.

Installation of these interfaces is extremely simple. The programmer simply invokes an ALS Prolog program called Access Expert which questions the programmer. The invocation is a simple as this:

```
>dbpro accexp
ALS Prolog Version 1.0x [1000]     DB Version
        Copyright (c) 1986  Applied Logic Systems, Inc.

?-accexp.
```

The dialog with Access Expert beins here ....

The first question is the name of the DBMS (here, dBaseIII), and the name of the data file containing the target relation (here, PARENT.DBF). Then 'access_expert' obtains all the information it needs to construct a basic interface by reading the header of the dBaseIII data file PARENT.DBF. It also asks whether there are any dBaseIII indexing files associated with PARENT.DBF. Next, it asks the name of the corresponding Prolog predicate (here, 'parent'), and finally it asks for the name of a file (say, PARENT.PRO) in which it should store the interface code it will generate. (This code is entirely Prolog code; a few very low-level supporting routines written in C have been added.) `access_expert' then generates Prolog code defining the interface and stores it in PARENT.PRO. All the programmer need to now to interface his Prolog program to PARENT.DBF is to load the file PARENT.PRO. The code in this file makes it appear that 'parent' consists of a collection of 5,000 facts corresponding to the 5,000 tuples in PARENT.DBF.

As an example, consider a simple relational database containing gross employment data by country and economic activity. The source data might originally have taken the following form:

Country	Economic Activity	PGNP	NPE	PT
Brazil				
	Community, Social & Personal Serv.	44	10,706	24
	Manufacturing	23	6,832	15
	Wholesale & Retail Trade etc.	12	4,276	10
Great Britain				
	Community, Social & Personal Serv.	19	6,200	27
	Manufacturing	28	7,155	31
	Wholesale & Retail Trade etc.	10	2,806	12

PGNP = % of GNP produced
NPE = Number of people employed
PT = % of total

As is common in the implementation of such databases, the actual entries are highly compressed or coded versions of the source data. In this case, the actual table might look like this:

```
Brazl   CSP   44    10706   24
Brazl   Man   23     6823   15
Brazl   WRT   12     4276   10
 ...    ...
GrBrt   CSP   19     6200   27
GrBrt   Man   28     7155   31
GrBrt   WRT   10     2806   12
 ...    ...
```

Let us assume that the file containing this table is named ECON.DBF. Moreover, let's also assume that dBaseIII was instructed to index this table on the first two columns; the files for these indicies will be named ECON1.NDX and ECON2.NDX. If we were to create the equivalent set of Prolog facts, the collection would look like this:

```
econ_data('Brazl', 'CSP', 44, 10706, 24).
econ_data('Brazl', 'Man', 23,  6823, 15).
econ_data('Brazl', 'WRT', 12,  4276, 10).
 ...     ...
econ_data('GrBrt', 'CSP', 19,  6200, 27).
econ_data('GrBrt', 'Man', 28,  7155, 31).
econ_data('GrBrt', 'WRT', 10,  2806, 12).
 ...     ...
```

However, using the ALS Prolog <=> Database Interface tools, you doesn't create these facts in your Prolog program. Instead, you invoke the 'access_expert' program and participate in a dialog similar to the following (where the user's responses are shown in **outline font**):

```
What is the name of the relation (data file)? econ
What is the database system you are using? dBaseIII
Do any of the fields have index files? yes

Fields in econ:
1.        Cntry characater  6
2.        Acvty characater  3
3.        PGNP  numeric     3
4.        NPE   numeric     7
5.        PT    numeric     3

Type in a list of numbers for the fileds that are indexed:   [1,2]

What is the name of the index file for Cntry? econ1
What is the name of the index file for Acvty? econ2
Will you need to read and write full records (y/n) ?   y
Do you need individual access to any of the fileds (y/n) ? n
What do you want the accessing Prolog predicate to be named? economic_data
What file should the interface code be stored in? econ

Interface code has been stored in the file econ.pro

Thanks for the work....
```

At this point, the access_expert stores the interface code (which the expert writes in Prolog with some calls on builtin functions which have been coded in C) in the file ECON.PRO. The data can now be accessed from ALS Prolog as if the facts listed above had been created as part of an ordinary Prolog program. All that needs to be done is load the file ECON.PRO along with any other Prolog files which will use these "facts". This is illustrated as follows:

```
>alspro econ
ALS Prolog 1.0 [nnnn]
Copyright (c) 1986 Applied Logic Systems, Inc.

?-economic_data('Brazl', 'Man', PGNP_Val, NPE_Val, PT_Val).
PGNP_Val = 23
NPE_Val = 6823
PT_Val = 15

yes.
```

One can build ALS Prolog programs over this data as if the data were expressed as ordinary Prolog facts. For example, suppose the file ECONANLY.PRO contains the following clause:

```
max_pe(Country, Activity)   :-
  setof( x(NPE, Where),  economic_data(Where, Activity, _, NPE, _),  List),
    List = [ x(_, Country) | _ ].
```

Assuming the database only contains data on Great Britain and Brazil, the following interaction would occur:

```
>alspro econ econanyl
ALS Prolog 1.0 [nnnn]
      Copyright (c) 1986 Applied Logic Systems, Inc.

?-max_pe(Country, 'Man').
Country = 'GrBrt'
yes.
```

An arbitrary number of data files can be accessed from one program in this manner. The files need not all be mentioned on the command line invoking ALS Prolog. Instead, 'consult' statements loading them can be placed in the primary program files. For example, the file ECONANYL.PRO might begin:

```
    :-consult(econ).
    max_pe(Country, Activity) :- ...
```

The command line would simply be:

```
         >alspro econanyl
```

Alternatively, one can simply invoke ALS Prolog and then load the files:

```
    >alspro
    ALS Prolog 1.0 [1000]
          Copyright (c) 1986 Applied Logic Systems, Inc.

    ?-consult(econanyl).
    Loading econanyl.pro...econanyl.pro loaded.
    ?-
```

The complexity of the analytic programs accessing relational database data in this manner is limited only by the imagination and energy of the programmer. magination and energy of the programmer. magination and energy of the programmer.

ALS has designed an extremely flexible and easy to use tool which constructs virtual interfaces between Prolog and relational databases. The present development version of the tool constructs interfaces between ALS Prolog and databases built with Ashton- Tate's dBaseIII database management system (DBMS). However, the tool is readily adaptable to relational databases built using other DBMS products.

Appendix 2
Technical Description of the Prolog/dBaseIII Inter ace

Introduction

The current interface between Prolog and dBaseIII has three parts to it. At tl · first or uppermost level, a user or application Prolog program sees a normal Prolo; relation which seems to be a large collection of variable-free facts. In actuality, the Prc ɔg facts do not exist in the Prolog program. Instead, the collection of facts is a phantom : virtual collection which is determined (at runtime) by a dBaseIII database. At this level the user or Prolog program has no idea of where the information actually resides. The r ·xt level (the second or middle level) consists of a collection of Prolog predicates wh: h know what a dBaseIII data file looks like, and which retrieve or store information fr m these database files as needed. The predicates at the first or uppermost level are d fined in terms of these middle level predicates. Finally, the third or lowest level is a coll ction of evaluatable predicates added to an ordinary Prolog system to do work associ: ed with reading or writing an arbitrary disk file. Each of these levels will be explained .n detail below.

In the discussion to follow, we will use a ficticious database

supply(Item,PartNo,NoInStock),

where:

> **Item** is a character field of length 20,
> **PartNo** is a part number of type numeric with length 10 bytes, includin; 2 to the right of the decimal, and
> **NoInStock** is the number of the items to be found in stock, of length 0 bytes with 0 digits to the right of the decimal point.

This makes the full length of a record 41 bytes, when the byte used by dBa: · III for marking a record as deleted or not is included.

Level 1: The User Application Program Level.

The user or application program, (generically referred to as the user), sees a ıtabase call as a normal Prolog goal. For instance, if the user wants to talk about the **supply** relation, he would first load a file called **supply**. Toexecute a query on this dat: ›ase, he would simply make a call such as

supply(Item, PartNo, NoInStock)

where the arguments would be either variables or constants for Prolog to unify a ıinst. If the user wishes to add new records (or tuples) to a database, a **dbAssert** call is n essary. In the **supply** example, the call would be

dbAssert(supply(Item,PartNo,NoInStock))

where the arguments would be the values for **supply** being entered.

The user is unaware if the goal is actually found in the Prolog clause database, or is off on disk in a database file. This information is in the **supply** file, which might either be actual Prolog clauses containg the data, or clauses describing the access path to the file on disk (here a dBase III data file) where the data actually resides. The reason for this design decision was to ensure that the user does not have to pay attention to the details of where a relation physically resides when writing the program or phrasing a query. This also means that a fully integrated database could be split across several database systems (such as R:Base, dBaseII, dBaseIII, or across a network channel). Only the creator of the database need know how the database was created. The user would simply load a file specific to the relation, and start to compute with it. Note that the user (or program) can simultaneously make use of many different database access predicates. They might all be specific to one DBMS system, or spread over a number of such systems.

As currently implemented, **dbAssert** is more dependent on the user knowing where the data resides, since it is not a normal Prolog **assert** call. However, the application program could be written entirely with normal **asserts**. Determining whether or not the relation is found in a database or not handled in one of two ways:

(1) Preprocessing an application program relative to the interface files before it is compiled would be one way. Here, the interface files created by the database designer would specify which predicates were are dBaseIII databases and which were not. Everywhere in the application Program file that an **assert** is seen which involves a dBaseIII file, the preprocessor would replace the **assert** call with a **dbAssert** call.

(2) Another possibility would be to construct a modified **assert**, which would notice which predicates corresponded to dBase III database items, and call **dbAssert** instead.

Level 2: Prolog Interface Level

As stated earlier, the user sees the data reading interface to the **supply** database as a call to

supply(Item,PartNo,NoInStock),

and the data writing interface as a call to

dbAssert(supply(Item,PartNo,NoInStock)).

The procedures invoked to execute each of these calls lie in the second tier of the interface. These second tier procedures are written in Prolog to make them easily modifiable. They contain knowledge of what the database actually looks like on the disk. In the case of this project, the DBMS managing these databases is dBaseIII. To interface to a new DBMS, these routines must be rewritten for the particular database.

A relation with no index files associated with any of its fields has a read interface clause of the following:

Head :- access(FileInfo, Head)

where **Head** is of the form

RelName(Arg1, ..., Argn)

where **RelName** is the name of the n-argument relation to be accessed by this predicate, and **FileInfo** is a term of the form

fileInfo(RelName, RecordInFile, SizeOfRecord, SizeOfHeader)

where **RecordInFile** is the record number of the record that the **access** predicate will read next. The starting value for **RecordInFile** should be -1, which means that no records have been read yet, and that the file is not open. A non-negative value means that the file has been opened, and the value gives the number of records read. This is a highly non-logical operation and should not be seen by the typical Prolog programmer. **SizeOfRecord** is the total number of bytes taken up by a given record. **SizeOfHeader** is the size of the header at the front of the dBase III datafile containing the information for **RelName**. For example,

```
supply(Item, PartNo, NoInStock)
    :-
    access(fileInfo(supply, -1, 41, 130),
    supply(Item, PartNo, NoInStock)).
```

access reads in data from the database by unifying the data it reads from the database against the particular instantiations of the arguments used in the call, backtracking if necessary. For instance, a goal of

```
supply('gas can',123,Number)
```

would call

```
access(fileInfo(supply,-1,41,130), supply('gas can',123,Number))
```

This goal would read records from the database, starting at the first record and moving sequentially through the database, until it found a record with the first argument being **'gas can'** and the second argument being 123. **Number** would then be unified with the number of gas cans in stock.

access gets the information from the data file by a call to

```
getInfo(RelName(Arg1,...,Argn))
```

whose purpose is to know the types of data found in a single record and how to read them. For instance, the above **access** call would call

```
getInfo(supply('gas can',123,Number))
```

and **getInfo** would fail if the data for the current record did not unify with the arguments of **supply**. **access** would then try the next record, failing if there were no more records to be found in the database.

getInfo has the form

```
getInfo(RelName(Arg1,...,Argn))
        :-
        DeleteByteCheck,
        CallForArg1,...,
        CallForArgn
```

where **CallForArgi** is a call that can read the particular data type for argument i, and **DeleteByteCheck** is **getChars(" ",1)**, since non-deleted records begin with a space. For dBaseIII, these fields and the associated calls that read them are as follows:

Character: **getChars(String,Length,Justification,Pad)**
Read a **String** of **Length** characters from the current datafile. This string will have a justification of either **left** or **right**, and the ASCII code of the pad character is **Pad**.

Numeric: **getNum(Number,Length,Decimal,Justification,Pad)**
Read **Length** characters from the current datafile, and convert them into a Prolog **Number** with **Decimal** digits to the right of the decimal point. The other arguments are the same as for character.

Date: **getDate(Date)**
Read 8 characters from the current input file and convert them to a structure of the form MM/DD/YY, where MM, DD, and YY are the numbers of the month, day, and year, respectively.

Logical: **getChars([Logical],1)**
Simply read 1 character to be used as the logical value.

In our current example, the clause generated would be

```
getInfo(supply(Item,PartNo,NoInStock)
        :-
        getChars(" ",1),
        getChars(Item,20),
        getNum(PartNo,10,2),
        getNum(NoInStock,10,0).
```

Files with indexed fields are treated slightly differently. Instead of the **access** predicate being used, the retrieval clause has the form

```
Head :- indexedGet(RelName,Head,SizeOfHeader,SizeOfRecord)
```

The variables have the same meaning they had above. A clause of the form

```
didntGet(Head) :- access(FileInfo,Head)
```

is also included, where arguments have the same form as before. This clause is used if none of the indexed arguments are ground in a call to **indexedGet**. In our example, the clauses needed would be

```
supply(Item,PartNo,NoInStock)
        :-
        indexedGet(supply,supply(Item,PartNo,NoInStock),130,41).

didntget(supply(Item,PartNo,NoInStock)
        :-
        access(fileInfo(supply,-1,130,41),supply(Item,PartNo,NoInStock).
```

For **indexedGet** to work, another level two predicate must have a true instance for every database access predicate with indexed fields. This is the

```
indexing(RelName,ListIndexedArgs)
```

relation, where **RelName** is the name of the relation and **ListIndexedArgs** is a list of items looking like **Number : FileName**, where **Number** is the argument number of an indexed field in **RelName** and **FileName** is the name of the **.ndx** file associated with that index. For instance, if the **supply** database is indexed on the second argument only, in file **partno.ndx**, then the clause for **supply** would be

```
indexed(supply, [2 : partno] ).
```

If **supply** was called and the second argument was a variable, then the **didntGet** clause for **supply** would be called.

Writing to a database is handled through the **dbAssert** call, which has the form

```
dbAssert(Head)
        :-
        output(RelName,PosNumRecs,Head,SizeOfRecord,SizeOfHeader)
```

where **Head** is of the form

```
RelName(Arg1,...,Argn),
```

where:

RelName is the name of the n-argument relation to be accessed by this predicate,
SizeOfRecord is the total number of bytes taken up by a given record,
SizeOfHeader is the size of the header at the front of the datafile containing the information for **RelName**, and
PosNumRecs is the byte position in the datafile where the current number of records in the database is stored.

For example,

```
dbAssertt(supply(Item,PartNo,NoInStock))
        :-
        output(supply,4,,supply(Item,PartNo,NoInStock),41,130).
```

Before **dbAssert** can be called, the file **supply.dbf** must be opened with **openRel**. After the **dbAsserts** are completed, **supply.dbf** must be closed with **closeRel**.

The predicate **output** writes data to the database. For instance, a goal of

```
dbAssert(supply('gas can',123,12))
```

would call

```
output(supply,4,supply('gas can',123,12),41,130)
```

which would write the record to the database.

output writes the information to the data file by a call to

```
writeInfo(RelName(Arg1,...,Argn))
```

whose purpose is to know the types of data found in a single record and how to write them. For instance, the above **output** call would call

```
writeInfo(supply('gas can',123,12))
```

and **writeInfo** would write out the converted Prolog data structures to the current output file. **writeInfo** has the form

```
writeInfo(RelName(Arg1,...,Argn))
        :-
        DeleteByteWrite,
        CallForArg1,...,
        CallForArgn
```

where **CallForArgi** is a call that can read the particular data type for argument i, and **DeleteByteWrite** is **writeChars(" ",1)**. This is necessary since non-deleted records begin with a space. For dBaseIII, these fields and the associated calls that read them are as follows:

Character: **writeChars(String,Length,Justification,Pad)**
Write a **String** of **Length** characters to the current datafile. This string will have a justification of either **left** or **right**, and the ASCII code of the pad character is **Pad**.

Numeric: **writeNum(Number,Length,Decimal,Justification,Pad)**
Write **Length** characters to the current datafile, after converting them from a Prolog **Number** with **Decimal** digits to the right of the decimal point. The other arguments are the same as for character.

Date: **writeDate(Date)**
Write 8 characters from the current input file, converting them from a structure of the form MM/DD/YY, where MM, DD, and YY are the numbers of the month, day, and year, respectively.

Logical: **writeChars([Logical],1)**
Simply write 1 character to be used as the logical value.

In our current example, the clause generated would be

```
writeInfo(supply(Item,PartNo,NoInStock)
        :-
        writeChars(" ",1),
        writeChars(Item,20),
        writeNum(PartNo,10,2),
        writeNum(NoInStock,10,0).
```

Files with indexed fields are treated slightly differently. Instead of the **output** predicate being used, the asserting clause has the form

```
dbAssert(Head,RecNo)
```

```
        :-
        indexOutput(RelName,Head,SizeOfRecord,SizeOfHeader,RecNo)
```

where the variables have the same meaning as above. The only addition is **RecNo**, which is the number of the record when it is entered in the main data file. Also, the

```
indexing(RelName,ListIndexedArgs)
```

relation must have an entry for **RelName**, where the format for **indexing** is the same as before. In our example, the clauses needed would be

```
dbAssert(supply(Item,PartNo,NoInStock),RecNo)
    :-
        indexOutput(supply,4,supply(Item,PartNo,NoInStock),41,130,RecNo).

indexed(supply,[2:partno]).
```

For detailed descriptions of how these routines are written, the reader is referred to the code, which has a large amount of documentation explaning how it works.

Level 3: The Evaluatable Predicates

For the two upper levels to work properly, some extra evaluatable predicates had to be added to Prolog. As much of the code as possible was written in Prolog to make experimentaion easy, but not everything could be written in Prolog. Predicates of this order include opening and closing of files, being able to move around arbitrarily in a file, and being able to read and write characters and numbers from and to a file. These will now be explained in detail.

openRel(RelName,Extension)
Opens thefile with name **RelName** and extension **Extension**. Once the file is open, the 'current datafile' will be set to this file. Consequently, all I/O will be from and to this file.

Once the current datafile has been set by either an **openRel** or a **changeRel**, the application program can operate on this file without further regard to the ID of the file. However, to operate on a different file, either an **openRel** or a **changeRel** must be issued. For instance,

```
?- openRel(supply,dbf).
```

would open up the file **supply.dbf** and set the current datafile to point to **supply.dbf**. If the application program then wants to talk to the file **partno.ndx**, a

```
?- openRel(partno,ndx).
```

must be entered. To go back to **supply.dbf**, the program would then issue a

```
?- changeRel(supply,dbf).
```

which would allow operations to be done to **supply.dbf**. When done with these files, a

```
?- closeRel(supply,dbf).
```

and a

?- closeRel(partno,ndx).

must b issued.

To ıove in the current datafile without having to read or write characters requires the **move bs** and the **moveRel** predicates. **moveAbs(Position)** will move to byte position **Positic** ı, while **moveRel(Rel)** will move **Rel** bytes from the current position in the file, whethe **Rel** be positive or negative. To find out the current byte position of the read/write pointer in the current datafile, **posRel(Position)** will unify the current position of the read/w te head with **Position. endRel(Position)** will unify **Position** with what it thinks is the byt position of the end of the file. **setEndRel(End)** will set the end of file position for the cur ent datafile to **End**. This is done whenever the application program adds any charact rs to the file. The beginning byte position of a file is 0.

Wri ng characters to the file requires using **writeChars(String,SizeString)**, where **String** is a Pr og list of ASCII codes for the value of the string, and **SizeString** is the length of **String**. For instance,

?- writeChars("hello, world",12).

would rite **hello, world** out to the current datafile. Reading characters is achieved through **getCha s(String,SizeString)**, where **SizeString** bytes will be read from the file, starting at the curr nt position of the read head, and stored in the Prolog list **String** in ASCII.

Sp cial routines are required to read and write *ints* (16 bit integers), *longs* (32 bit integer), and *doubles* (8 byte floating point) from a file, since these are sometimes stored in files ı their binary representations. These routines are:

getInt(Int),
writeInt(Int),
getLong(Long),
writeLong(Long),
getDouble(Double), and
writeDouble(Double).

The routines were all that were deemed necessary to be evaluatable predicates in Phase I. Parts of level 2 will most likely be moved into level 3 for speed considerations during Phase I

Appendix 3. Using the Access Expert

Thi: section describes how to use the access expert program to construct an inte face between Prolog and a given dBaseIII table.

Ste 1: Determine the name of the dBaseIII data file holding the table whi h contains the tuple which are to be represented as Prolog facts. In this exai :ple, we will call it **mytable.dbf**.

Ste 2: Determine the number of columns in the table. (Use dBaseIII if nec ssary to obtain this information.). We will suppose that the table in **my able.dbf** contains N columns.

Ste 3: Determine which columns of the table have had indexing files con: .ructed for them, and determine the names of those indexing files. For this discussion, we will assume that the following columns have the indicated inde :ing files associated with them:

Column Number	Indexing File Name
n_1	**indx1.ndx**
n_2	**indx2.ndx**

Ste 4: Decide on the name you want for the Prolog predicate which will sup y access to the facts corresponding to the tuples in the dBaseIII table. In t .is discussion, we will call the predicate **mypred**.

Ste 5: Determine whether you will be only reading the tuples in **my able.dbf**, whether you will only be writing new tuples back into **myt ıble.dbf**, or both.

Ste 6: Determine whether you will only need access (reading or writing) to s me of the fields in the tuples of **mypred.dbf**, or whether you will be acce sing the entire tuples (for either reading or writing).

Ste 7 Invoke the special version of ALS Prolog called 'dbpro' with 'acc xp' as a command-line argument. When ALS Prolog prompts you with its ' -' prompt, type 'accexp.' followed by return. This will appear as follows:

```
>dbpro accexp    type return here
ALS-Prolog Version 1.0x (DB Version)  [1000]
        Copyright (c) 1986  Applied Logic Systems
?-accexp.   type return here
```

Ste 8: The Access_Expert will greet you with its banner and then being aski g you questions. Your answers will be based on the information you gath :ed in steps 1-7. (In the following, what Access_Expert types is shown in (dinary font and your responses are shown in **outline font**.) The bann r typed by Access_Expert looks like this:

```
ACCESS EXPERT   An ALS DBTool
Copyright (c) 1986  Applied Logic Systems, Inc.
```

This DBTool will help the user create an access file tailored
for the particular database and application.

We will consider each question in turn. Your answer to each question must be terminated with a period followed by a return. (You can type 'help' to any of the Access Expert's questions. It will give you a brief explanation of what it wants.)

8.1:

What is the name of the relation (data file)? **mytable.**

As the example shows, just answer with the name of the data file, but don't include the extension (i.e., the '.dbf' in the case of dBaseIII).

8.2:

What is the database system you are using? **dBaseIII.**

At the moment, the only database system that Access Expert knows about is dBase III. However, this will change in the future. At this point in the dialog, Access Expert opens the file 'mytable.dbf' and reads the header information, learning the names of the columns, the types and sizes of the the column entries, etc.

8.3:

Do any of the fields of the relation have index files? **yes.**

If no fields have index files, just answer 'no' . Access Expert will then skip to question 8.7. If you answer yes, Access Expert prints a table of all the column names for the relation, the types of the entries in the columns, and their sizes. The table might look something like the following:

```
Fields in econ:
1.     column #1 name   column #1 type   column #1 size
2.     column #2 name   column #2 type   column #2 size
...    ......           .....            .....
N.     column #N name   column #N type   column #N size
```

8.4:

Type in a list of numbers for the fileds that are indexed: **[n1,n2].**

Access expert has to ask for this indexing information (as well as that which follows) becuase dBaseIII has no master data dictionary in which all such information is stored. In this question, simply type the list of numbers corresponding to the columns which have index files. Then Access Expert will follow with a sequence of questions asking for the names of the associated indexing files:

8.5:

What is the name of the index file for Cntry? **'indx1.ads'.**

8.6:

What is the name of the index file for Acvty? **'indx2.ads.'**

8.7:

Will you need to read and write full records (y/n) ? **y.**

8.8:

Do you need individual access to any of the fields (y/n) ? **n.**

If you do need individual access to any of the fields, answer y. Access Expert will ask you to identify the fields for which you require access.

8.9:

Your name for the accessing Prolog predicate? **my_pred.**

8.10:

File to store the interface code ? **'my_pred.pro'.**

Interface code has been stored in the file econ.pro

Thanks for the work....

At this point the interface code has been written into the file 'my_pred.pro'.

8.11: Using the interface is now quite easy. The file 'my_pred.pro' simply has to be loaded, either alone (for simply direct querying) or along with the applications Prolog program which will use 'my_pred'. For example, if 'my_prog.pro' is a file containing Prolog code for an applications program utilizing 'my_pred', the commands to load the files would be:

```
>dbpro my_pred my_prog
```

Alternatively, they can be consulted once inside ALS Prolog:

```
>dbpro
ALS Prolog 1.0x [1000]    DB Version
      Copyright (c) 1986 Applied Logic Systems, Inc.

?-[my_pred, my_prog].
Loadingmy_pred.pro...my_pred.pro  loaded.
Loadingmy_prog.pro...my_prog.pro  loaded.
?-
```

Appendix 5. The Graphics Interface Experiments

Introduction

The Graph package is a series of routines written in C for talking to a graphics output device, in this case the USI MultiDisplay Adapter Card, which looks like an IBM Color Graphics Adapter (CGA). This package is written using the ALS C interface for adding new evaluatable predicates to the ALS Prolog system. The main purpose for these routines is to show how Prolog could be used for doing graphical operations.

Discussion

Several C and Prolog routines were written to handle primitive 2-D graphics for the CGA. These include points and line drawing primitives, as well as color and coordinate transformation primitives.

To use the package, the user must first consult the file **graph.pro** and type

?- **graphInit.**

which will load the C graphics routines and initialize them for use. This initialization comprises of resetting the coordinate transforms. If at any time, the user is unhappy with the coordinate transforms and wants to reset them, he can type

?- **initGraph.**

which will only reset the coordinates. **graphInit** should only be typed once, since it loads the C routines.

The use must now decide which resolution he wishes to plot at, and tell Prolog which graphics mode to enter. To enter graphics mode in the lower resolution, type

?- **lowRes.**

which gives 200x320 pixels, and

?- **highRes.**

to get 200x640 pixels. The high resolution is available only in black and white, while the low resolution has 4 colors, which 4 depending on which color palette is chosen. To re-enter text mode on the CGA,

?- **textMode.**

will return the monitor to the text mode which allows 4 colors for letters.

Once in either graphics mode, the screen has a coordinate axis put on it where the x coordinates determine the row on the screen, and y determines the column. The point (0,0) is defined to be the center of the screen, while (-1,1), (1,1), (1,-1), and (-1,-1) define the upper left hand, upper right hand, lower right hand, and lower left hand corners respectively. Anything that is plotted outside of these limits will not show up on the screen. This does not mean, however, that any points input outside of the screen cannot be plotted. That is the purpose of the coordinate transforms, which will be discussed later.

Once in graphics mode, a color must be chosen for the pen to draw in. This color is chosen with the call **setColor(Color)**, which will set the pen to be **Color**. The colors which are available at a given time depend on which palette is chosen, which is set by the **palette(PaletteNumber)** call. Palette 0 has colors *black*, *white*, *magenta*, and *cyan*, while palette 1 has colors *black*, *green*, *red*, and *brown*. Once a palette has been chosen, as long as the user wishes a color in that palette, another **palette(PaletteNumber)** call is not necessary. Mixing of colors between palettes is not possible, because of hardware limitations.

To plot a point, type

?- point(X,Y).

which will put a point on the screen at coordinates (X,Y).

?- line(X1,Y1,X2,Y2).

will draw a line from point (X1,Y1) to (X2,Y2), while

?- lineTo(X,Y).

will draw a line from the last point plotted by either a **point**, **line**, or **lineTo** call, to point (X,Y). For instance,

?- point(1,1),lineTo(1,-1),lineTo(-1,-1),lineTo(-1,1),lineTo(1,1).

will draw a box on the screen.

Coordinate transforms are available for transforming input coordinates before they are plotted. They are scaling, translating, and rotating. These transforms are accumulated as they are received, but are independent from other types of transformations. For instance, scaling by 0.5, and then scaling by 0.5 again means the total scaling is now 0.25. The independence means that, for example, scaling will not affect a rotation or translation. An example of these routines would be

?- scale(0.5,0.3).

which would scale x coordinates by 0.5, and y coordinates by 0.3. Typing in the box example from above will show how this looks. Then,

?- translate(0.1,0.2).

moves the box 0.1 units in the x direction and 0.2 units in the y direction.

?- rotate(45).

then rotates the transformed square by 45 degrees. This rotation is done as if the square was still centered on the screen. Typing

?- initGraph.

resets the transformations to 0 for translation and rotation, and 1 for scaling.

Other available calls are **cls**, and **backGround(Color)**, which will clear the screen, and set the background color of the screen to **Color**, respectively. The legal values for **Color** for the background are *black*, *blue*, *green*, *cyan*, *red*, *magenta*, *brown*, *lightGrey*, *darkGrey*, *lightBlue*, *lightGreen*, *lightCyan*, *lightRed*, *lightMagenta*, *yellow*, and *white*.

An example program

An example program which uses the Graph package is found in **plot.pro**. The code will not be described here, the reader is refered to the code for that. What will be discussed here is how to use the **plot** program.

plot is for plotting mathematical equations on the screen. To use it, the user should type

?- [graph,plot], graphInit.

which will load the graphics package, initialize it, and consult **plot.pro** .

To use **plot**, the user would type

?- plot(Function,X,Start,Stop,Increment).

where **Function** is a univariant function in **X**, and the plot will be in the interval **[Start,Stop]**, by steps of size **Increment**. For instance,

?- plot(cos(x),x,-1,1,0.1).

would plot the function cos(x) in the interval [-1,1] in steps of 0.1. The user can set the color of the plot, or transform it in any way wished as described above.

To plot the same function, but change the interval and step size used for plotting,

?- setPlotLimits(Start,Stop,Step).

will change these values. Typing

?- replot.

will then draw the equation again.

There is also a predicate **axis**, which will draw an axis with tick marks on the screen. To use it, simply type

?- axis.

and it will appear on the screen.

END
7-87
DTIC